INTRODUCING CUDA 8

Pascal Support
New Architecture, Stacked Memory, NVLINK

Unified Memory
Simple Parallel Programming with large virtual memory

Libraries
nvGRAPH - library for accelerating graph analytics apps
FP16 computation to boost Deep Learning workloads

Developer Tools
Critical Path Analysis to speed overall app tuning
OpenACC profiling to optimize directive performance
Single GPU debugging on Pascal
INTRODUCING TESLA P100
New GPU Architecture to Enable the World’s Fastest Compute Node

- Pascal Architecture: Highest Compute Performance
- NVLink: GPU Interconnect for Maximum Scalability
- HBM2 Stacked Memory: Unifying Compute & Memory in Single Package
- Page Migration Engine: Simple Parallel Programming with 512 TB of Virtual Memory
UNIFIED MEMORY
UNIFIED MEMORY
Dramatically Lower Developer Effort

CUDA 6+

Kepler GPU

CPU

Unified Memory

Allocate Up To GPU Memory Size

Simpler Programming & Memory Model
Single allocation, single pointer, accessible anywhere
Eliminate need for explicit copy
Greatly simplifies code porting

Performance Through Data Locality
Migrate data to accessing processor
Guarantee global coherence
Still allows explicit hand tuning
void sortfile(FILE *fp, int N) {
    char *data;
    data = (char *)malloc(N);
    fread(data, 1, N, fp);
    qsort(data, N, 1, compare);
    use_data(data);
    free(data);
}

void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
GREAT PERFORMANCE WITH UNIFIED MEMORY

RAJA: Portable C++ Framework for parallel-for style programming

RAJA uses Unified Memory for heterogeneous array allocations

Parallel forall loops run on device

“Excellent performance considering this is a "generic" version of LULESH with no architecture-specific tuning.”

-Jeff Keasler, LLNL

LULESH Throughput

<table>
<thead>
<tr>
<th>Mesh size</th>
<th>Million elements per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>45^3</td>
<td>8.0x</td>
</tr>
<tr>
<td>100^3</td>
<td>1.9x</td>
</tr>
<tr>
<td>150^3</td>
<td>2.0x</td>
</tr>
</tbody>
</table>

GPU: NVIDIA Tesla K40, CPU: Intel Haswell E5-2650 v3 @ 2.30GHz, single socket 10-core
CUDA 8: UNIFIED MEMORY
Large datasets, simple programming, High Performance

CUDA 8

Pascal GPU

CPU

Unified Memory

Allocate Beyond GPU Memory Size

Enable Large Data Models
Oversubscribe GPU memory
Allocate up to system memory size

Simpler Data Access
CPU/GPU Data coherence
Unified memory atomic operations

Tune Unified Memory
Usage hints via cudaMemAdvise API
Performance
Explicit prefetching API
__global__
void setValue(int *ptr, int index, int val)
{
    ptr[index] = val;
}

void foo(int size) {
    char *data;
    cudaMemcpyManaged(&data, size);
    memset(data, 0, size);
    setValue<<<...>>>(data, size/2, 5);
    cudaDeviceSynchronize();
    useData(data);
    cudaFree(data);
}
HOW UNIFIED MEMORY WORKS IN CUDA 6

Servicing CPU page faults

---

**GPU Code**

```c
__global__
void setValue(char *ptr, int index, char val)
{
    ptr[index] = val;
}
```

**CPU Code**

```c
cudaMallocManaged(&array, size);
memset(array, size);
setValue<<<...>>>(array, size/2, 5);
```

---

**GPU Memory Mapping**

**CPU Memory Mapping**

---

Page Fault

Interconnect
HOW UNIFIED MEMORY WORKS ON PASCAL
Servicing CPU and GPU Page Faults

GPU Code

```c
__global__
Void setValue(char *ptr, int index, char val)
{
    ptr[index] = val;
}
```

CPU Code

```c
cudaMallocManaged(&array, size);
memset(array, size);
setValue<<<...>>>(array, size/2, 5);
```
USE CASE: ON-DEMAND PAGING

Graph Algorithms

Performance over GPU directly accessing host memory (zero-copy)

Unified Memory speed-up over system memory
- Blue: baseline
- Red: optimized

GPU working set / Total graph size

Large Data Set

Baseline: migrate on first touch
Optimized: best placement in memory
void foo() {
    // Assume GPU has 16 GB memory
    // Allocate 32 GB
    char *data;
    size_t size = 32*1024*1024*1024;
    cudaMallocManaged(&data, size);
}

32 GB allocation
Pascal supports allocations where only a subset of pages reside on GPU. Pages can be migrated to the GPU when “hot”.

Fails on Kepler/Maxwell
GPU OVERSUBSCRIPTION

Now possible with Pascal

Many domains would benefit from GPU memory oversubscription:

- **Combustion** - many species to solve for
- **Quantum chemistry** - larger systems
- **Ray tracing** - larger scenes to render
GPU OVERSUBSCRIPTION
HPGMG: high-performance multi-grid

Unified Memory oversubscription in AMR multi-grid codes

*Tesla P100 performance is very early modelling results
UNIFIED MEMORY ON PASCAL
Concurrent CPU/GPU access to managed memory

```c
__global__ void mykernel(char *data) {
    data[1] = 'g';
}

void foo() {
    char *data;
    cudaMallocManaged(&data, 2);

    mykernel<<<...>>>(data);
    // no synchronize here
    data[0] = 'c';

    cudaFree(data);
}
```

OK on Pascal: just a page fault

Concurrent CPU access to ‘data’ on previous GPUs caused a fatal segmentation fault
UNIFIED MEMORY ON PASCAL

System-Wide Atomics

```c
__global__ void mykernel(int *addr) {
    atomicAdd(addr, 10);
}

void foo() {
    int *addr;
    cudaMallocManaged(addr, 4);
    *addr = 0;

    mykernel<<<...>>>(addr);
    __sync_fetch_and_add(addr, 10);
}
```

Pascal enables system-wide atomics
- Direct support of atomics over NVLink
- Software-assisted over PCIe

System-wide atomics not available on Kepler / Maxwell
PERFORMANCE TUNING ON PASCAL

Explicit Memory Hints and Prefetching

Advise runtime on known memory access behaviors with `cudaMemAdvise()`

- `cudaMemAdviseSetReadMostly`: Specify read duplication
- `cudaMemAdviseSetPreferredLocation`: Suggest best location
- `cudaMemAdviseSetAccessedBy`: Initialize a mapping

Explicit prefetching with `cudaMemPrefetchAsync(ptr, length, destDevice, stream)`

- Unified Memory alternative to `cudaMemcopyAsync`
- Asynchronous operation that follows CUDA stream semantics

To Learn More:
S6216 “The Future of Unified Memory” by Nikolay Sakharnykh
Tuesday, 4pm
GRAPH ANALYTICS
GRAPH ANALYTICS
Insight from Connections in Big Data

... and much more: Parallel Computing, Recommender Systems, Fraud Detection, Voice Recognition, Text Understanding, Search
nvGRAPH
Accelerated Graph Analytics

Process graphs with up to 2.5 Billion edges on a single GPU (24GB M40)

Accelerate a wide range of applications:

<table>
<thead>
<tr>
<th>PageRank</th>
<th>Single Source Shortest Path</th>
<th>Single Source Widest Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>Robotic Path Planning</td>
<td>IP Routing</td>
</tr>
<tr>
<td>Recommendation Engines</td>
<td>Power Network Planning</td>
<td>Chip Design / EDA</td>
</tr>
<tr>
<td>Social Ad Placement</td>
<td>Logistics &amp; Supply Chain Planning</td>
<td>Traffic sensitive routing</td>
</tr>
</tbody>
</table>

nvGRAPH: 4x Speedup

48 Core Xeon E5
nvGRAPH on K40

PageRank on Wikipedia 84 M link dataset

developer.nvidia.com/nvgraph
Dependence Analysis

Easily Find the Critical Kernel To Optimize

The longest running kernel is not always the most critical optimization target.
Dependency Analysis

Visual Profiler

Unguided Analysis

Generating critical path

Dependency Analysis

Functions on critical path
DEPENDENCY ANALYSIS

Visual Profiler

APIs, GPU activities not in critical path are greyed out.
MORE CUDA 8 PROFILER FEATURES

Unified Memory Profiling

CPU Profiling

NVLink Topology and Bandwidth profiling

OpenACC Profiling
COMPILER IMPROVEMENTS
2X FASTER COMPILe TIME ON CUDA 8

NVCC Speedups on CUDA 8

- Average total compile times (per translation unit)
- Intel Core i7-3930K (6-cores) @ 3.2GHz
- CentOS x86_64 Linux release 7.1.1503 (Core) with GCC 4.8.3 20140911
- GPU target architecture sm_52

Performance may vary based on OS and software versions, and motherboard configuration.
HETEROGENEOUS C++ LAMBDA
Combined CPU/GPU lambda functions

```cpp
__global__ template <typename F, typename T>
void apply(F function, T *ptr) {
    *ptr = function(ptr);
}

int main(void) {
    float *x;
    cudaMallocManaged(&x, 2);

    auto square = 
        [=] __host__ __device__ (float x) { return x*x; };

    apply<<<1, 1>>>(square, &x[0]);
    ptr[1] = square(&x[1]);
    cudaFree(x);
}
```

Call lambda from device code

__host__ __device__ lambda

Pass lambda to CUDA kernel

... or call it from host code

Experimental feature in CUDA 8.
`nvcc --expt-extended-lambda`
HETEROGENEOUS C++ LAMBDA
Usage with Thrust

void saxpy(float *x, float *y, float a, int N) {
    using namespace thrust;
    auto r = counting_iterator(0);

    auto lambda = [=] __host__ __device__ (int i) {
        y[i] = a * x[i] + y[i];
    };

    if(N > gpuThreshold)
        for_each(device, r, r+N, lambda);
    else
        for_each(host, r, r+N, lambda);
}
BEYOND
FUTURE: UNIFIED SYSTEM ALLOCATOR

Allocate unified memory using standard malloc

CUDA 8 Code with System Allocator

```c
void sortfile(FILE *fp, int N) {
    char *data;

    // Allocate memory using any standard allocator
    data = (char *) malloc(N * sizeof(char));

    fread(data, 1, N, fp);

    qsort<<<...>>>(data,N,1,compare);

    use_data(data);

    // Free the allocated memory
    free(data);
}
```

- Removes CUDA specific allocator restrictions
- Data movement is transparently handled
- Requires operating system support
COOPERATIVE GROUPS
A Programming Model for Coordinating Groups of Threads

Support clean composition across software boundaries (e.g. Libraries)

Optimize for hardware fast-path using safe, flexible synchronization

A programming model that can scale from Kepler to future platforms
COOPERATIVE GROUPS SUMMARY
Flexible, Explicit Synchronization

Thread groups are explicit objects in the program

```c
thread_group group = this_thread_block();
```

Collectives, such as barriers, operate on thread groups

```c
sync(group);
```

New groups are constructed by partitioning existing groups

```c
thread_group tiled_partition(thread_group base, int size);
```
MOTIVATING EXAMPLE

Optimizing for Warp Size

```c
__device__
int warp_reduce(int val) {
    extern __shared__ int smem[];
    const int tid = threadIdx.x;

    #pragma unroll
    for (int i = warpSize/2; i > 0; i /= 2) {
        smem[tid] = val;  __syncthreads();
        val += smem[tid ^ i]; __syncthreads();
    }
    return val;
}
```

`__syncthreads()` is too expensive when sharing is only within warps
MOTIVATING EXAMPLE
Implicit Warp-Synchronous Programming is Tempting...

__device__
int warp_reduce(int val) {
    extern __shared__ int smem[];
    const int tid = threadIdx.x;

    #pragma unroll
    for (int i = warpSize/2; i > 0; i /= 2) {
        smem[tid] = val;
        val += smem[tid ^ i];
    }
    return val;
}
MOTIVATING EXAMPLE
Safe, Explicit Programming for Performance

Approximately equal performance to unsafe warp programming

```c
__device__
int warp_reduce(int val) {
    extern __shared__ int smem[];
    const int tid = threadIdx.x;

    #pragma unroll
    for (int i = warpSize/2; i > 0; i /= 2) {
        smem[tid] = val; sync(this_warp());
        val += smem[tid ^ i]; sync(this_warp());
    }
    return val;
}
```

Safe and Fast!
PASCAL: MULTI-BLOCK COOPERATIVE GROUPS

Provide a new launch mechanism for multi-block groups

Cooperative Groups collective operations like `sync(group)` work across all threads in the group

Save bandwidth and latency compared to multi-kernel approach required on Kepler GPUs

---

Normal `__syncthreads()`

Multi-block Sync
CUDA 8 AND BEYOND

JOIN THE CONVERSATION
#GTC16  

http://parallelforall.com

mharris@nvidia.com
@harrism